Tight Bounds for Online Balanced Partitioning in the Generalized Learning Model*

Harald Räcke¹, Stefan Schmid², and Ruslan Zabrodin¹

¹Technical University of Munich ²Technical University of Berlin

Resource allocation in distributed and networked systems such as the Cloud is becoming increasingly flexible, allowing these systems to dynamically adjust toward the workloads they serve, in a demand-aware manner.

Online balanced partitioning is a fundamental optimization problem underlying such self-adjusting systems. We are given a set of ℓ servers. On each server we can schedule up to k processes simultaneously (the capacity); overall there are $n=k\cdot\ell$ processes. The demand is described as a sequence of requests $\sigma_t=\{p_i,p_j\}$, which means that the two processes $p_i,p_j\in P$ communicate. Whenever an algorithm learns about a new request, it is allowed to move processes from one server to another, which however costs 1 unit per process move. If the processes (the endpoints of the request) are on different servers, it further incurs a communication cost of 1 unit for this request. The objective is to minimize the competitive ratio: the cost of serving such a request sequence compared to the cost incurred by an optimal offline algorithm.

Henzinger et al. (at SIGMETRICS'2019) introduced a learning variant of this problem where the cost of an online algorithm is compared to the cost of a static offline algorithm that does not perform any communication, but which simply learns the communication graph and keeps the discovered connected components together. This problem variant was recently also studied at SODA'2021.

In this paper, we consider a more general learning model (i.e., stronger adversary), where the offline algorithm is not restricted to keep connected components together. Our main contribution are tight bounds for this problem. In particular, we present two deterministic online algorithms: (1) an online algorithm with competitive ratio $\mathcal{O}(\max(\sqrt{k\ell \log k}, \ell \log k))$ and augmentation $1 + \epsilon$; (2) an online algorithm with competitive ratio $\mathcal{O}(\sqrt{k})$ and augmentation $2 + \epsilon$. We further present lower bounds showing optimality of these bounds.

^{*}Research supported by the German Research Foundation (DFG), grants 47002938 (FlexNet) and SPP 2378 (ReNO).

1. Introduction

The performance of many distributed applications (e.g., deep learning models such as GPT-4) critically depends on the performance of the underlying communication networks (e.g., during distributed training) [1,18]. Especially large flows (also known as elephant flows) may consume significant network resources if communicated across multiple hops; resources which would otherwise be available for additional flows [12,17]. A particularly innovative approach to improve the communication efficiency is to adjust the network resources dynamically (e.g., using virtualization), in a demand-aware manner: by moving frequently communicating vertices (e.g., processes or virtual machines) topologically closer (e.g., collocating them on the same server), transmissions over the network can be reduced.

When and how to collocate vertices however is an algorithmically challenging problem, as it introduces a tradeoff: as moving a vertex to a different server comes with overheads, it should not be performed too frequently and only when the moving cost can be amortized by the more efficient communication later. Devising good migration strategies is particularly difficult in the realm of online algorithms and competitive analysis, where the demand is not known ahead of time.

A fundamental algorithmic problem underlying such self-adjusting infrastructures is known as online balanced (graph) partitioning, which has recently been studied intensively [2,3,5,6,11, 13,14,19], see also the recent SIGACT News article on the problem [16]. In a nutshell (details will follow), in this model, we are given a set of ℓ servers, each of capacity k, which means that we can schedule up to k processes simultaneously on each server. There are $n = k \cdot \ell$ processes in total. We need to serve a sequence of communication requests $\sigma_t = \{p_i, p_j\}$ with $p_i, p_j \in P$ between processes. The sequence is revealed one-by-one to an online algorithm. Upon a new request, the algorithm is allowed to move processes from one server to another, which costs 1 unit per process move. It then needs to serve the request: if the processes (the endpoints of the request) are on different servers, it further incurs a communication cost of 1 unit for this request. The objective is to minimize the competitive ratio: the cost of serving such a request sequence compared to the cost incurred by an optimal offline algorithm.

In this paper we are interested in the learning variant of this problem, initially introduced by Henzinger et al. [14] at SIGMETRICS'2019 and later also studied at SODA'2021 [13]. In this variant, the cost of an online algorithm is compared to the cost of a static offline algorithm that is not allowed to perform any communication. Rather, requests must induce a communication graph which can be perfectly partitioned among the servers: if we define a demand graph G_{σ} over the set of processes and edges (p_i, p_j) for every request between p_i and p_j within σ , then the connected components of G_{σ} can be mapped to servers so that there is no inter-server edge.

For this setting, it is known that no online algorithm achieving a low competitive ratio can exist without resource augmentation: there is a lower bound of $\Omega(n)$ for any deterministic online algorithm [2,6]. In fact, currently the best known upper bound for this problem variant is nearly quadratic, namely $O(n^{23/12})$ [9].

1.1. Our Contributions and Novelty

We revisit the online graph partitioning problem in the learning model. Compared to prior work, we consider a more powerful adversary which is not restricted to keep processes on the same server once they communicated; rather, algorithms are allowed to communicate across servers. This model is not only more practical but also motivated as a next step toward solving the general online balanced partitioning problem. Since the lower bound of $\Omega(n)$ for algorithms

without resource augmentation of course also holds in this more general adversarial model, we in this paper study models with augmentation, as usual in the related work.

Our main contribution are intriguing tight bounds for this problem. We first show that the more general model is significantly more challenging as the lower bounds increase substantially. In particular, we show that even under arbitrary resource augmentation and even when compared to a static offline algorithm, any deterministic online algorithm has a competitive ratio of at least $\Omega(\sqrt{k})$.

Theorem 1.1. For any deterministic online algorithm ALG with unlimited resource augmentation, there exists a request sequence σ , such that $ALG(\sigma) \geq \Omega(\sqrt{k}) \cdot OPT_S(\sigma)$, where OPT_S is the optimum static offline algorithm.

The proof is based on a key observation regarding a scenario with two sets of processes A and B that initially are located on distinct servers. Given any deterministic algorithm ALG we construct a request sequence such that ALG performs poorly on requests between processes from $A \cup B$.

We then consider scenarios with only small resource augmentation and show an even higher lower bound, which again even holds when compared to a static offline algorithm:

Theorem 1.2. For any $k \geq 10$, any constant $\epsilon \leq \frac{1}{10}$, and any deterministic algorithm ALG with augmentation $1+\epsilon$, there is a request sequence σ , such that $ALG(\sigma) \geq \Omega(\max(\sqrt{k\ell \log k}, \ell \log k)) \cdot OPT_S(\sigma)$, where OPT_S is the optimum static offline algorithm.

Also this lower bound shows that the more general learning model is challenging. In [13], it was shown that in the standard learning model deterministic algorithms with resource augmentation $(1+\epsilon)$ have a competitive ratio of at least $\Omega(\ell \log k)$, but also an upper bound of $\mathcal{O}(\ell \log k)$ has been derived.

The proof of this lower bound is fairly technical and builds upon ideas of our simpler lower bound above as well as techniques from the lower bound in [13].

We then present almost optimal online algorithms which match our lower bounds. In our approach, we formulate the generalized learning problem as a linear program. We maintain a dual solution to the LP and bound the cost of our algorithm in terms of the dual profit. For this we reformulate the generalized learning problem as a graph problem: for a given request sequence we define a time-expanded graph. A key novelty of our approach is that this time-expanded graph is used in an online setting, where dual solutions are computed on the fly. Our online algorithms then consist of two parts, a clustering part and a scheduling part. The clustering part maintains a partition of the process set into disjoint pieces (clusters). The scheduling part of an algorithm then decides which cluster is mapped to which server, however, without splitting clusters across different servers.

Based on these algorithmic insights, we first show that very little augmentation (namely $1+\epsilon$) is sufficient to achieve a competitive ratio of $\mathcal{O}(\max(\sqrt{k\ell \log k}, \ell \log k))$. In particular, we prove the following theorem:

Theorem 1.3. There exists a deterministic online algorithm for the generalized learning problem with competitive ratio $\mathcal{O}(\max(\sqrt{k\ell \log k}, \ell \log k))$ and augmentation $1 + \epsilon$.

This algorithm uses our flow clustering procedure and uses the algorithm from the standard learning model presented in [13] as a subroutine in a second phase. In the second phase, hence entire connected components are assigned to a single server. However, we will show that the cost incurred during this phase is not significantly higher than that of the first phase.

We then present a second online algorithm which uses slightly more augmentation of $2 + \epsilon$ to achieve a significantly better competitive ratio of $\mathcal{O}(\sqrt{k})$, hence removing the linear factor ℓ , which also matches our corresponding lower bound. Also this algorithm relies on our flow clustering procedure, however, since our goal is to achieve a $\mathcal{O}(\sqrt{k})$ competitive ratio, we cannot simply apply the procedure as before. Concretely, we derive the following theorem:

Theorem 1.4. There exists a deterministic online algorithm for the general learning problem with competitive ratio $\mathcal{O}(\sqrt{k})$ and augmentation $2 + \epsilon$.

1.2. Further Related Work

The dynamic balanced (re-)partitioning problem was introduced by Avin et al. [2,6]. For a general online setting where requests can be arbitrary over time, the paper showed a linear in $\Omega(k)$ lower bound (even if the online algorithm is allowed significant resource augmentation), and also presented a deterministic online algorithm which achieves a competitive ratio of $O(k \log k)$ (with constant augmentation). That algorithm however relies on expensive repartitioning operations and has a super-polynomial runtime. Forner et al. [11] later showed that a competitive ratio of $O(k \log k)$ can also be achieved with a polynomial-time online algorithm which monitors the connectivity of communication requests over time, rather than the density. Pacut et al. [19] presented an $O(\ell)$ -competitive online algorithm for a scenario without resource augmentation and the case where k=3. The dynamic graph partitioning problem has also been studied in scenarios with less resource augmentation or where augmentation is even strictly forbidden. Regarding the former, Rajaraman and Wasim [22] presented a $O(k\ell \log k)$ -competitive algorithm for scenarios with only slight resource augmentation. Regarding scenarios without augmentation, Avin et al. [6] already presented a simple algorithm achieving a competitive ratio of $O(k^2 \cdot \ell^2)$, i.e., $O(n^2)$. Only recently a first online algorithm achieving a subquadratic competitive ratio of $O(n^{23/12})$ (ignoring polylog factors) has been presented [9]. An improved analysis of the original algorithm of Avin et al. was presented by Bienkowski et al. [8], by translating the problem to a system of linear integer equations and using Graver bases. The problem has further also been studied from an offline perspective by Räcke et al. who presented a polynomial-time $O(\log n)$ -approximation algorithm [20], using LP relaxation and Bartal's clustering algorithm to round it.

Deterministic online algorithms have also been studied for specific communication patterns, namely the ring. In [4,5], the adversary generates the communication sequence from an arbitrary (adversarially chosen) random distribution in an *i.i.d.* manner [4,5] from the ring. In this scenario, it has been shown that even deterministic algorithms can achieve a polylogarithmic competitive ratio. In [21], a more general setting is considered where the adversary can choose edges from the ring arbitrarily, in a worst-case manner. It is shown that a polylogarithmic competitive ratio of $O(\log^3 n)$ can be achieved in this case by randomized algorithms; against a static solution, the ratio can be improved to $O(\log^2 n)$ (this ratio is strict, i.e., without any additional additive terms). The problem on the ring is however very different from ours and the corresponding algorithms and techniques are not applicable in our setting.

The learning variant considered in this paper was introduced by Henzinger et al. [13,14]. The authors presented a deterministic exponential-time algorithm with competitive ratio $O(\ell \log \ell \log k)$ as well as a lower bound of $\Omega(\log k)$ on the competitive ratio of any deterministic online algorithm. While their derived bounds are tight for $\ell = O(1)$ servers, there remains a gap of factor $O(\ell \log \ell)$ between upper and lower bound for the scenario of $\ell = \omega(1)$. In [13], Henzinger et al. present deterministic and randomized algorithms which achieve (almost) tight bounds for the learning variant. In particular, a polynomial-time randomized algorithm is described which

achieves a polylogarithmic competitive ratio of $O(\log \ell + \log k)$; it is proved that no randomized online algorithm can achieve a lower competitive ratio. Their approach establishes and exploits a connection to generalized online scheduling, in particular, the works by Hochbaum and Shmoys [15] and Sanders et al. [24]. In our paper, we consider a more general variant of the learning model, where the algorithms are not restricted to keep process pairs, once they communicated, on the same server forever.

More generally, dynamic balanced partitioning is related to dynamic bin packing problems which allow for limited repacking [10]: this model can be seen as a variant of our problem where pieces (resp. items) can both be dynamically inserted and deleted, and it is also possible to open new servers (i.e., bins). The goal is to use only an (almost) minimal number of servers, and to minimize the number of piece (resp. item) moves. However, the techniques of [10] do not extend to our problem. Another closely related (but technically different) problem is online vertex recoloring (disengagement) [7,23]. This problem is the flip side of our model: in disengagement, the processes in requests should be are separated and put on different servers (rather than collocated).

1.3. Future Work

We see our work as a stepping stone toward solving the general online balanced partitioning problem. While we remove the restriction that the algorithm needs to keep communication pairs on the same server, we still require that the graph induced by the communication requests can be perfectly partitioned across the servers (i.e., without inter-server edges). In other words, while our paper removes the restriction on the adversary, it remains to remove this restriction on the input as well. In particular, it remains an open questions whether a polylogarithmic competitive ratio can be achieved in general setting by randomized algorithms.

2. Model and Preliminaries

We now present our formal model in details and also introduce the necessary preliminaries.

Let ℓ denote the number of servers and k the *capacity* of a server, i.e., the maximum number of processes that can be scheduled on a single machine. We use $V = \{p_1, p_2, \dots, p_n\}$ with $n \leq \ell k$ to denote the set of processes. In each time step t we receive a request $\sigma_t = \{p_i, p_j\}$ with $p_i, p_j \in V$, which means that these two processes communicate. For a request sequence σ we define the *demand graph* G_{σ} as a (multi-)graph that contains a vertex for every process and an edge (p_i, p_j) for every request between p_i and p_j within σ .

Serving a communication request costs exactly 1 if both requested processes are located on different servers, otherwise 0. We call this the *communication cost* of the request. Before the communication, an online algorithm may (additionally) decide to perform an arbitrary number of migrations. Each migration of a process to another server induces a cost of 1 and contributes to the *migration cost* of the request.

An assignment of processes to servers is referred to as a (current) configuration or scheduling of the algorithm. We start in the initial configuration \mathcal{I} where each server contains exactly k processes. The initial server of a process p is called the home server of p and denoted with h(p).

In the end, after performing all migrations, each server should obey its capacity constraint, i.e., it should have at most k processes assigned to it. The goal is to find an online scheduling of processes to servers for each time step that minimizes the sum of migration and communication costs and obeys the capacity constraints.

Resource augmentation. We usually consider online algorithms that do not have to fulfill the capacity constraints exactly but are allowed to have a little bit of slack. Whereas the optimum offline algorithm may schedule at most k processes on any server at any given time, the online algorithm may schedule αk processes on any server for some parameter $\alpha > 1$. We say the online algorithm uses resource augmentation α .

Restricting the request sequence. In the *learning variant* of the problem[13, 14] the request sequence is restricted so as to allow for a static solution that does not require any communication cost. This means there exists a very good solution (in terms of communication cost) and the goal is to *learn* this solution as the requests appear.

Henzinger et al. [14] introduced this learning variant and compared the cost of an online algorithm to the cost of a *static* offline algorithm that does not perform any communication. This means initially the optimum algorithm (which knows the request sequence in advance) chooses a static placement that does not require any communication. It then switches to this placement paying some migration cost, and then serves the request sequence without any further migration and/or communication.

In this paper we introduce the generalized learning model where we have the same restriction on the request sequence but we lift some of the restrictions on the optimal offline algorithm that we compare against. We consider two scenarios. In the first scenario the offline algorithm is static, i.e., it must initially migrate to a placement and then serve the request sequence without changing the placement of the processes. However, in contrast to the standard learning model [13] this placement does not need to be perfect in the sense that serving the request sequence does not incur communication.

In the second scenario the offline algorithms is completely dynamic and only has to obey capacity constraints on the servers.

Comparing static and dynamic adversaries. Let OPT_S and OPT denote the optimum static and dynamic offline algorithm, respectively. For a request sequence σ we use $OPT_S(\sigma)$ and $OPT(\sigma)$ to denote the cost of these algorithms on request sequence σ . We drop σ if it is clear from the context and, hence, use OPT_S and OPT to denote an algorithm and also its cost. The following simple fact holds because the dynamic algorithm is more powerful than its static counter-part.

Fact 2.1. For any request sequence σ we have $OPT_S(\sigma) \geq OPT(\sigma)$.

Next, we show that there exist request sequences for which the optimal static algorithm performs significantly worse than the optimal dynamic algorithm.

Lemma 2.2. For any k there exists a request sequence σ where $OPT_S \geq \Omega(k) \cdot OPT$. Moreover, this inequality holds even if OPT_S has unlimited resource augmentation whereas OPT has no resource augmentation at all.

Proof. We fix two servers s_p and s_q with p_1, \ldots, p_k initially residing on s_p , and q_1, \ldots, q_k initially residing on s_q . We generate the following request sequence of length $2\lfloor \frac{k}{2} \rfloor$:

$$\sigma = \{p_1, q_1\}, \{p_1, q_2\}, \dots, \{p_1, q_{\lfloor \frac{k}{2} \rfloor}\}, \{p_1, p_2\}, \{p_1, p_3\}, \dots, \{p_1, p_{\lfloor \frac{k}{2} \rfloor + 1}\}$$

Let C be the connected component in the demand graph G_{σ} . C contains $\lfloor \frac{k}{2} \rfloor$ processes of server s_p and $\lfloor \frac{k}{2} \rfloor$ processes of server s_q . Every optimal static algorithm (even with unlimited

resource augmentation) will either move $\Omega(k)$ processes or pay $\Omega(k)$ communication cost. However, a dynamic algorithm that first swaps p_1 and q_k , serves the first k-1 requests, swaps p_1 and q_k again, and serves the remaining requests, pays only a cost of 2 for the swaps.

The above lemma means that there may be a very large gap between the static and dynamic model. Nevertheless, we will show matching upper and lower bounds, where the lower bounds are against a static adversary and the upper bounds are against a dynamic adversary.

3. Lower Bounds

In [13], it was shown that in the standard learning model deterministic algorithms with resource augmentation $(1+\epsilon)$ have a competitive ratio of at least $\Omega(\ell \log k)$, and deterministic algorithms with resource augmentation $(2+\epsilon)$ have a competitive ratio of at least $\Omega(\log k)$. Note that the \mathcal{O} -notation is hiding dependencies on the parameter ϵ .

These lower bounds directly carry over to the generalized model since we only make the adversary more powerful. We now show that this change in the model makes an important difference because the lower bounds increase substantially.

3.1. Lower Bounds for Algorithms with Unlimited Augmentation

In this section we show a lower bound of $\Omega(\sqrt{k})$ even for the case with unlimited augmentation. This is in stark contrast to the standard learning model in the literature (which we generalize in this paper) where already an augmentation of $2 + \epsilon$ allowed an upper bound of $\mathcal{O}(\log k)$.

Theorem 1.1. For any deterministic online algorithm ALG with unlimited resource augmentation, there exists a request sequence σ , such that $ALG(\sigma) \geq \Omega(\sqrt{k}) \cdot OPT_S(\sigma)$, where OPT_S is the optimum static offline algorithm.

Proof. Fix a deterministic algorithm ALG and two arbitrary servers. Assume Server 1 contains processes $S_1 = \{p_1, p_2, \dots, p_{k/2}\}$, and Server 2 contains processes $S_2 = \{p_{k/2+1}, p_{k+2}, \dots, p_k\}$ (wlog. assume that k is even). As long as not all processes p_1, \dots, p_k are on the same server we issue a request of the form $\{p_i, p_{i+1}\}$, $i \in \{1, \dots, k-1\}$ for which p_i and p_{i+1} are on different servers. ALG incurs cost for each of these requests. Hence, its cost is at least $\max\{|\sigma|, k/2\}$ because it has to move k/2 processes in order to stop the sequence.

The optimum algorithm OPT identifies a pair (p_i, p_{i+1}) in the range $i \in \{k/2 - s, k/2 + s\}$ (s < k/2) that experiences the least number of requests. The range contains 2s + 1 pairs, and, hence, one of them has at most $|\sigma|/(2s+1) \le |\sigma|/(2s)$ requests. Regardless, of which pair attains the minimum, OPT can move at most 2s processes between Server 1 and Server 2 so that both servers have k processes and only requests for this minimum pair requires communication (this uses the fact that servers have enough dummy processes that are not involved in any requests and can be moved between servers so as to balance the load). Consequently, OPT can serve the sequence with cost $2s + |\sigma|/(2s)$. Choosing $s = \Theta(\sqrt{k})$ gives the theorem.

3.2. Lower Bounds for Algorithms with Augmentation $1+\epsilon$

For very small resource augmentation, in the standard learning model, there is a lower bound of $\Omega(\ell \log k)$ for any online algorithm, i.e., there is a strong dependency on the number of servers. In this section, we show that in the generalized learning model, a significantly higher lower bound of $\Omega(\max(\sqrt{k\ell \log k}, \ell \log k))$ can be shown.

The proof follows a structure similar to that presented in [13]. However, we enhance it by integrating key insights from both their approach and Theorem 1.1. This combination of ideas allows us to establish stronger lower bounds in the generalized learning model. See

Theorem 1.2. For any $k \geq 10$, any constant $\epsilon \leq \frac{1}{10}$, and any deterministic algorithm ALG with augmentation $1+\epsilon$, there is a request sequence σ , such that $ALG(\sigma) \geq \Omega(\max(\sqrt{k\ell \log k}, \ell \log k)) \cdot OPT_S(\sigma)$, where OPT_S is the optimum static offline algorithm.

In the following, we assume that $\epsilon \leq 1/10$ and $\epsilon k = 2^m$ for a positive integer m.

Notation Fix any deterministic algorithm ALG with augmentation $1 + \epsilon$. We construct a request sequence σ and analyze the costs $ALG(\sigma)$ and $OPT_S(\sigma)$. In the following we refer to requests as edges.

We associate each server with a color and each process with the color of its home server. We say, the $main\ server$ of a color c is the server that, out of all servers, currently contains the largest number of processes with color c (ties broken arbitrarily).

Each server reserves ϵk of its initial processes as padding processes. These processes will not be included in any request, so migrating them will not result in any communication cost for OPT_S. These processes are only necessary so that OPT_S can balance the load between servers and create a static offline solution without resource augmentation.

We construct the request sequence σ such that the connected components of the demand graph G_{σ} mostly consist of processes of the same color. In fact, there will only be one component, the *special component*, that contains different colors. The other non-padding processes will be part of *colored components*. At each point in time a color c has a rank r_c . All but (at most) one of its components will have the same size 2^{r_c} . We call a c-colored components that has size 2^{r_c} a regular component of color c. A c-colored component that does not have size 2^{r_c} is called the extra component for color c. We call processes in extra components extra processes and processes in regular components regular processes.

During the request sequence, we merge components by issuing a request between processes from different components. To start, we create the special component. Let $S_1 = \{p_1, \ldots, p_{3\epsilon k}\}$ be arbitrary $3\epsilon k$ processes initially stored on Server 1 and $S_2 = \{p_{3\epsilon k+1}, \ldots, p_{6\epsilon k}\}$ arbitrary $3\epsilon k$ processes initially stored on Server 2. We say processes p_i and p_{i+1} are consecutive processes. We merge S_1 and S_2 into the special component by issuing requests (p_i, p_{i+1}) for $i \in 1, \ldots, 6\epsilon k-1$. All other non-padding processes initially form regular components of size 1.

We will ensure that ALG stores each component on a single server by repeatedly requesting processes of that component that are stored on different servers. On the other hand, OPT_S is not restricted to storing each component on a single server. It will first perform some migrations and then stay in this configuration for the entire request sequence, possibly incurring some communication cost.

Most components of a color c will be scheduled on the main server for color c. We call a color c spread-out if the number of processes in c-colored components that are not located on the main server for c is at least ϵk (note that regardless of their color, special and padding processes are not considered to be in a c-colored component).

The request sequence. We generate the request sequence in iterations. In the first phase of each iteration, as long as there is a component C that is split across servers, we request a process pair from C that resides on different servers. If C is a regular component or an extra component, we choose this pair arbitrarily. However, if C is the special component, we choose an arbitrary pair (p_i, p_{i+1}) that is located on different servers.

The first phase of an iteration ends when ALG schedules all processes of any component entirely on the same server.

In the second phase, we proceed as follows: According to Lemma 3.1, there must exist a spread-out color c. Let r_c be its rank. We find a matching M_c between the regular components of color c that maximizes the number of edges that cross between servers. Then we merge the matched components. By construction, the size of the newly created components is 2^{r_c+1} . If the number of regular components is odd, there will be an unmatched component. We merge this component with the extra component. Before the merge, the size of the leftover component was less than 2^{r_c} ; thus, after the merge, its size will be less than 2^{r_c+1} . The rank of color c increases by one.

This finishes the iteration for constructing the request sequence. The construction finishes once the rank of each color is at least m.

Cost Analysis

Lemma 3.1. At the start of the second phase of an iteration, there exists a spread-out color.

Proof. Recall that a color c is spread-out if the number of processes in c-colored components that are not located on the main server for c is at least ϵk . A color can have at most $3\epsilon k$ special processes and has exactly ϵk padding processes. This means that it has at least $k-4\epsilon k$ processes in c-colored components.

Assume for contradiction that all colors have strictly more than $(1 - 5\epsilon)k$ non-special non-padding processes at their main server (these are the processes in *colored components*). Since $\epsilon \leq 1/10$, we have $(1 - 5\epsilon)k \geq k/2$. Thus, a server can be the main server for only one color, which means that there is a one-to-one mapping of colors to main servers.

Now, consider the color c, whose main server currently holds the special component. The size of the special component is $6\epsilon k$. Hence, there are at most $(1+\epsilon)k - 6\epsilon k = (1-5\epsilon)k$ non-special non-padding processes at this server. This gives a contradiction.

The next two lemmas derive bounds on the costs of ALG and OPT_S . Let R denote the number of requests generated due to the special component being split across servers.

Lemma 3.2. The cost of OPT_S on the request sequence σ is at most $12\min(\sqrt{R}, \epsilon k)$.

Proof. We analyze two cases:

1. $\sqrt{R} \le \epsilon k$:

OPT_S can proceed as follows. Within the special component it identifies an edge that is requested the fewest number of times and lies within the sub-sequence $p_{3\epsilon k-\sqrt{R}}, \dots, p_{3\epsilon k+\sqrt{R}}$ of processes. As the total number of (special) requests is R and the range contains at least \sqrt{R} edges there must exist an edge that is requested at most \sqrt{R} times. Let this be the edge (p_i, p_{i+1}) . OPT_S can now shift at most \sqrt{R} processes between server S_1 and S_2 so that processes p_1, \dots, p_i are on S_1 and processes $p_{i+1}, \dots, p_{6\epsilon k}$ are on server S_2 . It then uses the padding processes to rebalance the load of S_1 and S_2 (another movement of at most \sqrt{R} processes). Hence, it incurs a movement cost of $2\sqrt{R}$ and a communication cost of \sqrt{R} for the requests between p_i and p_{i+1} .

The remaining requests (non-special requests) are all between processes that are on the same server. Hence, OPT_S does not pay anything for these requests. This gives $OPT_S(\sigma) = 3\sqrt{R}$.

2. $\sqrt{R} > \epsilon k$:

Let c^* be the last color that reaches the rank of m in the sequence σ . OPT_S initially moves the special component to the c^* -colored server s. The number of non-special non-padding processes on s is at least $k - 3\epsilon k - \epsilon k \ge 6\epsilon k$, where the inequality holds because $\epsilon \le 1/10$. These are all processes in c^* -colored components.

Since the rank of c is m, the size of all c^* -colored components except the extra component is exactly $2^m = \epsilon k$. Since the size of the extra component is less than ϵk , there must exist at least six c^* -colored components of size exactly ϵk on s. We move three of these components to Server 1 and the other three to Server 2, so that every server contains exactly k processes. Notice that all requests are created between processes of the same component and no component is split between different servers. Thus, OPT pays only for the migration cost, and therefore its cost is at most $12\epsilon k$.

In summary, OPT_S pays at most $12 \min(\epsilon k, \sqrt{R})$.

Lemma 3.3. The online algorithm ALG has cost $ALG(\sigma) \ge R + \epsilon k\ell \log(\epsilon k)$.

Proof. We can divide the cost of $ALG(\sigma) = cost_s + cost_r$ into two parts: $cost_s$ represents the cost due to the requests within the special component, and $cost_r$ represents the cost due to the requests within regular components. Each request results in a cost of 1 for ALG, either due to migration or communication. Therefore, $cost_s \geq R$.

Now, we derive a lower bound on $\cos t_r$. Since we issue requests until ALG puts all processes of a component on a single server, ALG has to pay at least the size of the smaller component each time two components on different servers merge. Hence, if two such components of size 2^r merge, ALG pays at least 2^r .

According to Lemma 3.1 in each iteration we find a spread-out color c. Thus, there are at least $\epsilon k = 2^m$ processes of color c that are not on their main server. Let r be the rank of c. Since the size of the extra component is less than 2^r , there must exist 2^{m-r} regular c-colored components of size exactly 2^r that are not on their main server. We fix these 2^{m-r} components. Now, we greedily pick for each such component a unique matching partner that is not on the same server. Since for every component there are at least 2^{m-r} candidate partners, we can easily find such a matching. Thus, the size of the matching \mathcal{M}_c^r must be at least 2^{m-r} , and since each merge results in a cost of 2^r for ALG, its overall cost in every iteration is at least $2^{m-r}2^r = \epsilon k$.

There are ℓ colors, and during each iteration the rank of one color increases by one. The algorithm stops when the rank of each color is at least m. Therefore, there are at least $\ell m = \ell \log(\epsilon k)$ iterations. Hence, $\cot_{\mathbf{r}} \geq \epsilon k \ell \log(\epsilon k)$. Summarizing, $\mathrm{ALG} \geq R + \epsilon k \ell \log \epsilon k$.

Finally, we can state the theorem:

Theorem 1.2. For any $k \geq 10$, any constant $\epsilon \leq \frac{1}{10}$, and any deterministic algorithm ALG with augmentation $1+\epsilon$, there is a request sequence σ , such that $ALG(\sigma) \geq \Omega(\max(\sqrt{k\ell \log k}, \ell \log k)) \cdot OPT_S(\sigma)$, where OPT_S is the optimum static offline algorithm.

Proof. Let C be the competitive ratio of ALG. Due to Lemmas 3.2 and 3.3 we know that $C \geq (R + \epsilon k \ell \log(\epsilon k))/(12\min(\epsilon k, \sqrt{R}))$. We show in separate steps that $C \geq \frac{1}{12}\ell \log(\epsilon k)$ and $C \geq \frac{1}{12}\sqrt{\epsilon k \ell \log(\epsilon k)}$.

- a) Clearly, $C \ge (R + \epsilon k \ell \log \epsilon k)/(12\epsilon k) \ge \frac{1}{12}\ell \log \epsilon k$.
- b) Next, we analyze two subcases:

i) $R > \epsilon k \ell \log \epsilon k$. Then:

$$C \ge \frac{R + \epsilon k \ell \log \epsilon k}{12\sqrt{R}} \ge \frac{1}{12}\sqrt{R} > \frac{1}{12}\sqrt{\epsilon k \ell \log \epsilon k}$$

ii) $R \leq \epsilon k \ell \log \epsilon k$. Then:

$$C \ge \frac{R + \epsilon k\ell \log \epsilon k}{12\sqrt{R}} \ge \frac{\epsilon k\ell \log \epsilon k}{12\sqrt{R}} \ge \frac{1}{12}\sqrt{\epsilon k\ell \log \epsilon k}$$

The above two subcases imply that $C \ge \frac{1}{12} \sqrt{\epsilon k \ell \log \epsilon k}$.

The two cases directly imply the lemma.

4. Algorithmic Concepts

In this section, we develop basic algorithmic tools that will help us solve the generalized learning problem. Our approach involves formulating the generalized learning problem as a linear program. We maintain a dual solution to the LP and bound the cost of our algorithm in terms of the dual profit. For this we reformulate the generalized learning problem as a graph problem.

4.1. Time-Expanded Graph

For a given request sequence σ we define the *time-expanded graph* \bar{G}_{σ} as the follows. We introduce one node for every server $s \in S$, and $|\sigma|$ nodes $p^1, \ldots, p^{|\sigma|}$ for every process $p \in V$. The first set of nodes are called *server nodes* and the nodes p^i , $i \geq 1$ are called *process time nodes*. For simplicity we use p^0 for a process p to denote the server node that in the initial configuration holds process p (thus, a server node s in \bar{G}_{σ} may have many different names; at least one for every process initially located on s).

The edges of the undirected graph \bar{G}_{σ} are defined as follows. For every process p and time-step $t \in \{1, \ldots, |\sigma|\}$ we add an edge (p^{i-1}, p^i) . In addition, we add an edge (p^t, q^t) if the t-th request is between processes p and q. We call the first set of edges migration edges and the second set communication edges. Figure 1 illustrates the time-expanded graph.

4.2. Linear Program Formulation

Suppose we assign edge-lengths $d_e \in \{0,1\}$ to the edges of the graph \bar{G}_{σ} , with the following meaning: if a communication edge (p^t, q^t) has length 1, processes p and q may be scheduled on different servers at time t (then the request is served via communication). Similarly, if a migration edge $(p^{\tau-1}, p^{\tau})$ has length 1 the process p may switch its server at time τ (meaning servers for time $\tau - 1$ and τ are different).

With this interpretation a distance of 0 between a node p^{τ} and a server node s implies that p is scheduled on s at time τ . Consequently, the distance between any two server nodes s and s', must be at least 1, as otherwise the process time nodes on a shortest path between s and s' would have to be scheduled on both servers.

Let \mathcal{P}_S denote the set of simple paths in \bar{G}_{σ} that connect two different server nodes. The above discussion implies that LP Primal in Figure 2 gives a lower bound on $\mathrm{OPT}(\sigma)$.

We can interpret the dual linear program Dual in 2 as a flow problem. We define $c_e := \sum_{P \in \mathcal{P}_S: e \in P} f_P$ as the *congestion* of an edge e. Furthermore, we can interpret the variable f_P

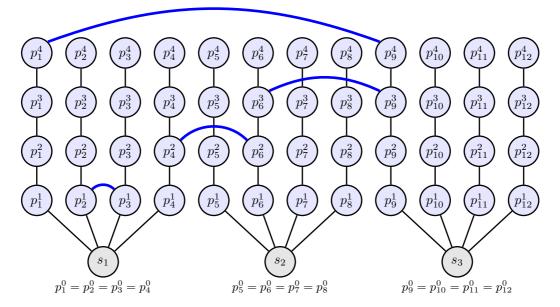


Figure 1: An illustration of the time expanded graph \bar{G}_{σ} for $\ell=3$ servers, server capacity k=4, and $|\sigma|=4$ requests. The bottom row are the server nodes, the remaining nodes are process time nodes. The request sequence is $\sigma=(\{p_2,p_3\},\{p_4,p_6\},\{p_6,p_9\},\{p_1,p_9\})$. The blue/bend/thick/horizontal edges are communication edges. All other edges are migration edges.

as the flow we send from one server to another on the path P. Therefore we call \mathcal{P}_S the set of flow-augmenting paths in the following. The objective is to maximize the overall flow amount while ensuring that the congestion of an edge is at most one.

Lemma 4.1. Let $\mathcal{P} \subseteq \mathcal{P}_S$ be a set of edge-disjoint simple paths. Then, $|\mathcal{P}| \leq \mathrm{OPT}(\sigma)$.

Proof. We set $f_P = 1$ for $P \in \mathcal{P}$, i.e., we send a flow amount of one on each such path. Since the paths are edge-disjoint, the congestion on an edge is at most 1. Thus, there is a feasible dual solution with objective value $|\mathcal{P}|$. Hence, the lemma follows from weak duality.

4.3. Clustering

We separate our online algorithms into two parts, a clustering part and a scheduling part. The clustering part maintains a partition of the process set V into disjoint pieces C_1, C_2, \ldots . We call this partition a clustering and the individual pieces clusters. The scheduling part of an algorithm then decides which cluster is going to which server but it does not split clusters among different servers. In the following we describe the flow clustering procedure, which operates on a subset $S \subseteq V$ of processes and decides how to split these subsets into clusters. This procedure forms a basic building block in our algorithms and is used for making clustering decisions when the optimum cost is rather low.

Flow Clustering Procedure Let $S \subseteq V$ denote a subset of processes. The flow procedure FLOW receives requests among processes in S and maintains a clustering S of S. The procedure stops

$$\begin{array}{lll} \underline{\text{LP Primal:}} & \underline{\text{LP Dual:}} \\ & & & & \\ & & & \\ & & & \\ &$$

Figure 2: Linear program relaxation of the generalized learning problem

when OPT must have had a large cost for requests in S.¹

Recall that we associate each server with a unique color. In addition to maintaining the clustering the flow procedure also assigns colors to clusters with the understanding that (usually) a cluster $C \in \mathcal{S}$ with color c is scheduled on the server with color c. However, the scheduling part may deviate from this "suggestion" in order to enforce balance constraint. Moreover, not every cluster has a color. There is one *special cluster* that is not assigned a color. All other clusters in \mathcal{S} have a color assigned and are called *colored clusters*.

Each request between processes in S might trigger an update of S. We (artificially) define the cost of the flow procedure for an update operation as the number of processes that change their color. Later we will show that the actual cost for scheduling the clusters is not too much more then this cost.

The procedure distinguishes between two types of processes in S: linked processes L that always stay within their initial cluster, and free processes F, which can be migrated between clusters. Initially, all processes in S are linked. During the procedure, linked processes may become free, but once free, a process may not become linked again. Crucially, the number of free processes (roughly) acts as a lower bound of an optimum solution for requests involving processes in S. The procedure is started with a cost parameter Z. Once the number of free processes reaches Z the procedure stops. Further requests involving a process from S are handled by another building block.

Let $\bar{G}[S]$ be the time-expanded graph induced by the process set S. To compare the update cost of the flow procedure to OPT the procedure maintains a set of disjoint paths $\mathcal{P} \subseteq \mathcal{P}_S$ in $\bar{G}[S]$. By setting $f_P = 1$ for $P \in \mathcal{P}$, we get a feasible dual solution to the linear program. In order to construct these paths we will prove the following invariant:

Invariant 4.2. Let τ be the current time-step. For each c-colored cluster C and each process $p \in C$, there is a path $Q_{\tau}(p)$, in the time-expanded graph $\bar{G}[S]$, that

- i) connects the server node with color c to p^{τ} (and does not contain other server nodes);
- ii) only contains nodes of the form v^i , $i \leq \tau$, $v \in C$;
- iii) exactly one of the processes that corresponds to nodes on the path is linked;
- iv) is edge disjoint from all paths in \mathcal{P} .

¹For now we assume that their are no requests of the form $\{s, x\}$, with $s \in S$ and $x \notin S$. Later in Section 6, we will have such requests. Then we will merge clustering procedures.

4.3.1. Request Processing

The procedure operates in iterations. At the beginning of an iteration, the free processes are part of the special cluster. A request $\sigma_t = (p, q)$ between processes p and q with $p, q \in S$ is handled as follows.

- 1. If p and q are in the same cluster, we ignore the request as it does not require any communication/migration.
- 2. Suppose, exactly one of p and q is in the special cluster. W.l.o.g., let q be the process in the special cluster. Then, p must belong to a colored cluster C. We migrate q to C.
- 3. Otherwise, both p and q belong to colored clusters. According to Lemma 4.3 there are paths $Q_t(p)$ and $Q_t(q)$, which fulfill the properties of Invariant 4.2. In particular, there are exactly two linked processes p' and q' that are contained in the paths $Q_t(p)$ and $Q_t(q)$, respectively.
 - We turn p' and q' into free processes.
 - We update the set \mathcal{P} by adding the path $P = Q_t(p) \circ (p^t, q^t) \circ Q_t^{-1}(q)$.
 - We move all free processes to the special cluster.
 - If now $|F| \geq Z$, the procedure terminates. Otherwise, we proceed to the next iteration.

4.3.2. Correctness Analysis

Lemma 4.3. The Invariant 4.2 is maintained throughout the execution of the procedure.

Proof. Fix a colored cluster C and a process $p \in C$. We first show that the invariant holds at the start of an iteration.

At this time all processes in C are linked, since the free processes are moved to the special cluster at the end of the previous iteration. Hence, p is linked. Whenever, a path is added to \mathcal{P} at the end of an iteration all linked processes that correspond to nodes on the path are converted into free processes (these are exactly two). Hence, the paths in \mathcal{P} cannot contain any node of the form q^i , $i \geq 1$ where q is linked (they can contain q^0 as this is a server node and exists under many different names).

Therefore, the path $p^0, p^1, \dots, p^{\tau}$ is edge-disjoint from any path in \mathcal{P} . All other constraints in Invariant 4.2 trivially hold for this path. This completes the proof for the start of an iteration.

Now, suppose that the time-step τ increases and a request $\sigma_{\tau} = \{x, y\}$ occurs (for the new value of τ). Let p denote a process in a colored cluster C and let s_c denote the server node with color c. By induction hypothesis there is a path P from s_c to $p^{\tau-1}$ that fulfills constraints ii,iii, and iv of Invariant 4.2. By extending P by the edge $p^{\tau-1}, p^{\tau}$ we get the desired path for p^{τ} .

It remains to construct a path for the case that a colored cluster C changes by adding a free process to it. For this suppose that for the request $\sigma^{\tau} = \{x, y\}$, $y \in C$ and x is a free process. Then, we already have constructed a path that connects s_c to y_{τ} . By extending this path via the edge (x^{τ}, y^{τ}) we obtain the desired path to x^{τ} . Clearly, the edge (x^{τ}, y^{τ}) cannot be contained in any path from \mathcal{P} as these paths only contain nodes with time-stamp strictly less than τ . \square

Lemma 4.4. The paths in \mathcal{P} are pairwise edge disjoint.

Proof. We prove the lemma by induction. Initially, the set \mathcal{P} is empty. Now, assume that the paths in the current set \mathcal{P} are pairwise disjoint and that the request $\sigma_{\tau} = \{p, q\}$ ends the current iteration, i.e., a new path $P = Q_{\tau}(p) \circ (p^{\tau}, v^{\tau}) \circ Q_{\tau}^{-1}(q)$ is added to \mathcal{P} .

By Lemma 4.3, the paths $Q_{\tau}(u)$ and $Q_{\tau}(v)$ are edge disjoint with any path in \mathcal{P} . Clearly, no path in \mathcal{P} can contain the edge (p^{τ}, q^{τ}) . Furthermore, the paths $Q_{\tau}(p)$ and $Q_{\tau}(q)$ cannot share edges, since p and q are in different colored clusters (Property ii of Invariant 4.2). Hence, the path P is a simple path from one server node to another, that is edge disjoint with any path in \mathcal{P} .

4.3.3. Cost Analysis

Let F be the current set of free processes of the FLOW-procedure, and let X be its special cluster.

Lemma 4.5. $|F| = 2|\mathcal{P}|$.

Proof. Initially, the sets F and \mathcal{P} are both empty. We change them only at the end of each iteration. We add two new processes to the set F, and one new path to \mathcal{P} . Hence, at all times $|F| = 2|\mathcal{P}|$ holds.

Lemma 4.6. At any time the update cost of the FLOW procedure is at most $|F|^2 - |X|$.

Proof. We prove the statement by using induction on the number of requests. Initially, F and X are both empty. Hence, the base case is trivial. For the induction step assume, that prior to the current request the cost is at most $|F|^2 - |X|$.

There are two cases where the procedure incurs cost and/or the sets F or X change.

• A request $\{x, p\}$ occurs between a process $x \in X$ and a process p from a colored cluster C:

Subsequently, x migrates to the colored cluster, which induces an update cost of 1. Since x leaves the special cluster X, its size decreases by one. Hence, the cost after this step is at most

$$|F|^2 - |X| + 1 = |F|^2 - (|X| - 1) = |F|^2 - |X'|$$
,

where X' is the new special cluster. The set F does not change.

• A request $\{p,q\}$ occurs between processes $p \in C_p$ and $q \in C_q$, where C_p, C_q are colored clusters:

In this case we create two new free processes and all free processes are moved to the special cluster. Let F' be the new set of free processes and X' the special cluster after this step. By construction, X' = F'. The cost of migration of the free processes is at most |F'|. Hence, the new update cost is at most

$$|F|^2 - |X| + |F'| = (|F'| - 2)^2 - |X| + |F'| = |F'|^2 - 4|F'| + 4 - |X| + |F'| \le |F'|^2 - |X'|.$$

The last inequality holds because $|F'| \ge 2$. Hence, the invariant is maintained.

5. A Deterministic $\mathcal{O}(\max(\sqrt{k\ell \log k}, \ell \log k))$ -competitive Algorithm with Augmentation $1 + \epsilon$

In this section, we introduce a deterministic algorithm that solves the generalized learning problem online with a competitive ratio of $\mathcal{O}(\max(\sqrt{k\ell \log k}, \ell \log k))$ and augmentation $1 + \epsilon$.

A key ingredient of our approach is the deterministic online algorithm for the standard learning model presented in [13], which we use as a subroutine LEARN.

5.1. Algorithm

Let \mathcal{I} denote the initial configuration, where each set $C \in \mathcal{I}$ corresponds to the processes initially located at some server. We assign each $C \in \mathcal{I}$ the color of its respective server. The algorithm for the general learning problem operates in two phases and consists of two main components: the FLOW procedure (4.3) and the LEARN subroutine.

1. Phase 1:

We execute the flow procedure with cost parameter $Z = \min(\sqrt{k\ell \log k}, \epsilon k)$ on the process set V, starting from the initial configuration \mathcal{I} , until it terminates. After processing each request, the flow procedure outputs a clustering \mathcal{S} , which consists of $|\mathcal{I}|$ colored clusters and one special cluster. The assignment of clusters to servers is as follows: the colored clusters are scheduled on their corresponding servers, while the special cluster is placed on Server 1.

2. Phase 2:

Once the flow procedure terminates, we reset the configuration to the initial state \mathcal{I} . We then reprocess the request sequence from the beginning using the LEARN algorithm.

Note that LEARN operates in the standard learning model, which means that it assigns entire connected components of the demand graph to a single server. However, we will demonstrate that the cost incured during the second phase is not significantly higher than that of the first phase.

5.2. Correctness Analysis

Lemma 5.1. The algorithm uses augmentation at most $1 + \epsilon$.

Proof. By construction of the flow procedure, only free processes are allowed to leave their initial cluster. The procedure ends if the number of free processes reaches $Z = \min(\sqrt{k\ell \log k}, \epsilon k)$. Therefore, at most $\min(\sqrt{k\ell \log k}, \epsilon k) \le \epsilon k$ processes will leave their initial cluster during the execution of the flow procedure. Since each cluster uniquely corresponds to a server, at most ϵk processes will leave their home server. Consequently, no server can be overloaded by more than ϵk processes. In the second phase, we apply the LEARN algorithm from [13], which by design, uses augmentation at most $1 + \epsilon$.

5.3. Cost Analysis

The following lemma is not explicitly stated in [13] but directly follows from their analysis. See Appendix A.1.

Lemma 5.2. [[13]] For any request sequence σ that obeys the learning restriction the cost LEARN(σ) is at most $\mathcal{O}(k\ell \log k)$.

Next, we show that the competitive ratio of the online algorithm is $\mathcal{O}(\max(\sqrt{k\ell \log k}, \ell \log k))$.

Lemma 5.3. For a request sequence σ the cost of the online algorithm is at most $\mathcal{O}(\max(\sqrt{k\ell \log k}, \ell \log k))$. OPT (σ) .

Proof. According to Lemma 4.6, at any point during the execution of the flow procedure, its cost is at most $|F|^2 - |X|$. This gives

$$FLOW(\sigma) \le |F|^2 - |X| \le 2|\mathcal{P}| \cdot Z \le 2Z \cdot OPT(\sigma)$$
,

where the second inequality holds due to Lemma 4.5 and the fact that $|F| \leq Z$ as the procedure terminates when |F| reaches Z. The final inequality follows because the paths in \mathcal{P} form a feasible dual solution to the linear program, and, hence, $OPT(\sigma) \geq |\mathcal{P}|$.

The cost for reverting the configuration to the initial state is at most the cost incurred in the first phase, i.e., at most $FLOW(\sigma)$.

The cost of the second phase is at most the cost of the LEARN-algorithm. According to Lemma 5.2 this is at most $ck\ell \log k$ for some constant c. When the flow procedure ends $\text{OPT}(\sigma) \geq |\mathcal{P}| \geq Z/2$. We now consider two cases:

• $\sqrt{k\ell \log k} \le \epsilon k$. Then

$$\mathtt{LEARN}(\sigma) = ck\ell \log k = c\sqrt{k\ell \log k} \sqrt{k\ell \log k} \leq 2c\sqrt{k\ell \log k} \cdot \mathsf{OPT}(\sigma) \enspace ,$$

because $OPT(\sigma) \ge \sqrt{k\ell \log k}/2$. Therefore, the total cost for both phases is at most

$$4Z \cdot \text{OPT}(\sigma) + 2c\sqrt{k\ell \log k} \cdot \text{OPT}(\sigma) = \mathcal{O}(\sqrt{k\ell \log k}) \cdot \text{OPT}(\sigma)$$
.

• $\epsilon k < \sqrt{k\ell \log k}$. In this case we have

$$\mathtt{LEARN}(\sigma) = ck\ell \log k = \frac{c}{\epsilon}\ell \log(k) \cdot \epsilon k \leq \frac{2c}{\epsilon}\ell \log(k) \cdot \mathtt{OPT}(\sigma) \ .$$

Therefore the total cost for both phases is at most

$$4Z \cdot \mathrm{OPT}(\sigma) + \frac{2c}{\epsilon} \ell \log(k) \cdot \mathrm{OPT}(\sigma) \le \mathcal{O}(\sqrt{k\ell \log k} + \ell \log k) \cdot \mathrm{OPT}(\sigma).$$

Combining both cases gives the lemma.

Lemmas 5.3 and 5.1 give the following theorem.

Theorem 1.3. There exists a deterministic online algorithm for the generalized learning problem with competitive ratio $\mathcal{O}(\max(\sqrt{k\ell \log k}, \ell \log k))$ and augmentation $1 + \epsilon$.

6. A Deterministic $\mathcal{O}(\sqrt{k})$ -competitive Algorithm with Augmentation $2+\epsilon$

In this section, we introduce an online algorithm that solves the generalized learning problem with augmentation $2 + \epsilon$ and achieves a competitive ratio of $\mathcal{O}(\sqrt{k})$. We build on the concepts discussed in the previous sections, particularly the FLOW procedure. However, since our goal is to achieve a $\mathcal{O}(\sqrt{k})$ competitive ratio, we cannot simply apply the procedure as before.

6.1. Overall Algorithm

The algorithm consists of two main components: The *Clustering algorithm* and the *Scheduling algorithm*.

- 1. The Clustering algorithm takes the request sequence as input and maintains a clustering S of the processes. When a new request is received, the algorithm may adjust the clustering, causing the processes to change their clusters.
- 2. The Scheduling algorithm takes a clustering S from the clustering algorithm and assigns the clusters to the servers. Changes in clustering may require some clusters to be rescheduled. Each cluster is always entirely scheduled on a single server.

6.2. Clustering Algorithm

The basic idea of the clustering algorithm is to group processes into a set of clusters S, by running multiple FLOW procedures (see Section 4.3), each on a connected component of the demand graph \bar{G}_{σ} .

6.2.1. Clustering Rules

Let C be the set of connected components of the current demand graph G_{σ} . We partition C into two sets C_S and C_L .

- For each $C \in \mathcal{C}_L$ we have that $C \in \mathcal{S}$, that means that C forms itself a cluster. We call C a large cluster.
- For each $C \in \mathcal{C}_S$ we maintain a FLOW procedure I(C) with cost parameter $Z = \sqrt{k}$, that operates on the connected component C. We call I(C) a procedure or instance and say that I(C) manages the component C. Each procedure I(C) partitions its connected component C into smaller sub-clusters \mathcal{S}_C . To avoid ambiguities, we call them pieces. A piece is either colored, or it is the special piece of its corresponding FLOW procedure. Furthermore, each procedure maintains a set of free processes $F_C \subseteq C$.

Assignment of pieces to clusters We say a piece S is δ -monochromatic, if at least $\delta |S|$ ($\delta \geq \frac{1}{2}$) processes in S share the same c-colored home server. The remaining processes in S are free. For every color c the clustering algorithm maintains a special cluster S_c , which we call the c-colored cluster. A piece S either forms a singleton cluster that only contains piece S, or it is assigned to the c-colored cluster, if c is the color of S. The assignment of pieces to clusters is done as follows.

All pieces initially consist of a single process, and thus belong to their corresponding ccolored cluster. Whenever a piece S changes we examine the new piece S'. Let c be the
color of S'.

- If S' is not $\frac{1}{2}$ -monochromatic, it becomes a singleton cluster.
- If S' is $\frac{3}{4}$ -monochromatic, it is assigned to the c-colored cluster.
- Otherwise, S' is assigned to the c-colored cluster iff S was assigned to this cluster. This means, that once a piece is assigned to a colored cluster, it will only be reassigned to a singleton cluster after a significant number of free processes have migrated to that piece.

Cost The cost of the clustering algorithm is defined as the total number of cluster changes made by processes. Each time a process is reassigned to a different cluster, it incurs a cost of 1.

During the execution of the algorithm, sets of processes are often *merged*. Let A and B be two disjoint sets of processes. When merging them into a single set $C = A \cup B$, the algorithm migrates all processes from the smaller set to the larger one. Afterward, the (now) empty set is deleted.

Notice, not all merges incur a cost. If both A and B are c-colored pieces that already reside within the same colored cluster, merging them does not involve any cost, as no process changes its cluster. On the other hand, if either A or B is a singleton cluster, the merge will incur a cost equivalent to $\min(|A|, |B|)$, reflecting the number of processes that need to change clusters.

6.2.2. Instance Management

Instance initialization The clustering algorithm starts by creating an instance $I(\{v\})$ for each process $v \in V$. Each set $\{v\}$ is assigned the color of v's home server. As a result, each process is initially contained in the c-colored cluster, if c is the color of its home server. Thus, initially $C_S = \{\{v\} \mid v \in V\}$ and $C_L = \emptyset$.

Instance merge Throughout the execution of the algorithm, whenever a request involves two different components $A, B \in \mathcal{C}_S$, we merge their corresponding instances, I(A) and I(B), into a single instance I(C), where $C = A \cup B$. Each instance maintains at most one c-colored piece and exactly one special piece. This invariant must be preserved during the merge. The merging process is implemented as follows:

- Colored pieces: For each color c, if there is a c-colored piece in both A and B (i.e., $C_A \in \mathcal{S}_A$ and $C_B \in \mathcal{S}_B$), these pieces are combined into a single piece $C_A \cup C_B$ in the new instance I(C).
- Special pieces: Let X_A and X_B be the special pieces of instances I(A) and I(B), respectively. These are combined into a single special piece $X_C = X_A \cup X_B$.
- Flow-augmenting paths and free processes: The set of flow-augmenting paths for the merged instance is set as $\mathcal{P}_C = \mathcal{P}_A \cup \mathcal{P}_B$, combining the paths from both original instances. Similarly, the set of free processes is set as $F_C = F_A \cup F_B$, uniting the free processes from A and B.

Instance stop A procedure continues to operate until the number of free processes in its component reaches \sqrt{k} , at which point the procedure stops. If a procedure I(C) stops, all its pieces (the whole connected component C) are merged into a singleton cluster. Consequently, C moves from C_S to C_L .

6.2.3. Request Processing

The requests are processed as follows. Let $\sigma_t = \{u, v\}$ be the current request.

If u and v belong to different connected components A and B, respectively, we first execute the component merge operation CompMerge(A, B). After this operation, it is guaranteed that u and v are part of the same connected component C.

If $C \in \mathcal{C}_L$, the request is served without any cost, since the algorithm ensures that u and v are in the same cluster. Otherwise, if $C \in \mathcal{C}_S$, the request is forwarded to the corresponding procedure I(C) for further processing.

CompMerge operation A component merge operation CompMerge (A, B) is performed as follows. Let \mathcal{C}'_S and \mathcal{C}'_L be the connected component sets after the CompMerge operation.

1. $A \in \mathcal{C}_L$ or $B \in \mathcal{C}_L$

Without loss of generality, assume $A \in \mathcal{C}_L$. If B is also a large cluster, we merge the two clusters. Thus, the new large component set becomes $\mathcal{C}'_L = \mathcal{C}_L - \{B, A\} + \{A \cup B\}$.

In case $B \in \mathcal{C}_S$, we stop the I(B) instance that manages B, and merge all pieces of I(B) (i.e., the entire component B) into cluster A. As a result, $\mathcal{C}'_S = \mathcal{C}_S - \{B\}$ and $\mathcal{C}'_L = \mathcal{C}_L - \{A\} + \{A \cup B\}$.

2. $A \in \mathcal{C}_S$ and $B \in \mathcal{C}_S$

In this case, there are instances I(A) and I(B) that manage A and B, respectively. We merge I(A) and I(B) into a single instance $I(A \cup B)$. The component sets are updated as follows: $\mathcal{C}_S' = \mathcal{C}_S - \{B, A\} + \{A \cup B\}$.

6.3. Scheduling Algorithm

The scheduling algorithm is taken directly from [21]. For completeness, we repeat the algorithm here.

The scheduling algorithm receives a set of clusters S from the clustering algorithm and maintains an assignment of the clusters to the servers in a way that keeps the load on each server within specified bounds. The scheduling algorithm assigns each c-colored cluster to the server s with the matching color c. All other clusters are scheduled on arbitrary servers.

Upon a new request the clustering algorithm could change existing clusters. Some processes might change their cluster, and some clusters might merge. This changes the size of clusters and, hence, the load distribution among the servers can become imbalanced. The scheduling procedure re-balances the distribution of clusters among servers such that there are at most $(2 + \epsilon)k$ processes on any server, for an $\epsilon > 0$.

Assume, that after the execution of the clustering algorithm server s has load greater than $(2 + \epsilon)k$. We perform the following re-balancing procedure. While server s has load greater than 2k, we take a non-colored cluster C in s and move it to a server s' with load at most k. Such a cluster C must exist, since the overall number of processes in colored clusters is at most 2k. Furthermore, such a server s' must also exist because the average load is at most k. Since $|C| \leq k$, s' has a load at most 2k afterwards.

6.4. Analysis

For each $C \in \mathcal{C}_S$, the procedure I(C) maintains two sets: the set of flow-augmenting paths \mathcal{P}_C and the set of free processes F_C . By construction of the FLOW procedure, each path $P \in \mathcal{P}_C$ contains only nodes of the form v^i for $i \geq 0$ and $v \in C$. Additionally, F_C is always a subset of C.

We extend these definitions to all components in \mathcal{C} . We define:

- \mathcal{P}_C : The set of flow-augmenting paths of procedures that have at any point managed C or any subset of C.
- F: The set of free processes across all procedures that have operated at any point in time.
- $F_C = F \cap C$, for a $C \subseteq V$.

Due to the design of the instance merge operation, for any $C \in \mathcal{C}_S$, these definitions naturally align with the sets of flow-augmenting paths and free processes maintained by the instance I(C).

Additionally, for each $C \in \mathcal{C}_S$, let X_C denote the special piece of the instance I(C).

Finally, let $\mathcal{P} = \bigcup_{C \in \mathcal{C}} \mathcal{P}_C$ represent the set of all flow-augmenting paths across all components. By Lemma 4.5 and by construction of the instance/component merge operation, $|\mathcal{P}| = \bigcup_{C \in \mathcal{C}} |\mathcal{P}_C| = \bigcup_{C \in \mathcal{C}} 2|F_C| = 2|F|$

6.4.1. Correctness Analysis

First, we prove the augmentation guarantees of the algorithm presented in this section.

Lemma 6.1. The algorithm uses at most augmentation $2 + \epsilon$.

Proof. Let C be a colored cluster. By construction, at most half of the processes in C are free. Since there are k processes that have a c-colored home server, the overall size of a colored cluster is bounded by 2k.

Each time the clustering algorithm modifies existing clusters and creates a load on a server s with more that $2+\epsilon$ processes, the scheduling algorithm re-balances this server by moving non-colored clusters, such that it has at most 2k processes. Such a re-balancing is always possible, since the size of a colored cluster is at most 2k. Thus, there is always a non-colored cluster that we can move away from s. This ensures that no server contains more than $2+\epsilon$ processes at any given time.

Next, we show that the paths in \mathcal{P} are pairwise edge-disjoint, which implies that there is a feasible dual solution to the linear program 2 with cost $|\mathcal{P}|$.

Lemma 6.2. For two connected components $A, B \in \mathcal{C}$, the paths in $\mathcal{P}_A \cup \mathcal{P}_B$ are pairwise edge disjoint.

Proof. By design of the FLOW procedure, a path $P_A \in \mathcal{P}_A$ contains only nodes of the form v^i for $i \geq 0$ and $v \in A$, and a path $P_B \in \mathcal{P}_B$ contains only nodes of the form u^i for $i \geq 0$ and $u \in B$. Since $A \cap B = \emptyset$, P_A and P_B cannot share an edge.

Lemma 6.3. The Invariant 4.2 is maintained after a procedure merge.

Proof. Assume, that Invariant 4.2 holds prior to a merge of two procedures I(A) and I(B) into procedure I(C), $C = A \cup B$. We show that the invariant is preserved after the merge.

W.l.o.g. let $v \in A$ be a process that belongs to a colored piece J and let $Q_{\tau}(v)$ be the path that fulfills the properties of Invariant 4.2 prior to the merge. By construction, v also belongs to some colored piece J' after the merge. We show, that $Q_{\tau}(v)$ still fulfills the properties of Invariant 4.2. Properties i) and iii) still trivially hold, since the path remains unchanged. By design, $J \subseteq J'$, thus, property ii) is also trivially maintained.

It remains to show that $Q_t(v)$ is edge disjoint with the flow-augmenting paths in $\mathcal{P}_C = \mathcal{P}_A \cup \mathcal{P}_B$. Since the invariant holds prior to the merge, $Q_t(v)$ is edge disjoint with paths in \mathcal{P}_A . Furthermore, by design, a path $P \in \mathcal{P}_B$ contains only nodes of the form u^i for $i \geq 0$ and $u \in B$. The path $Q_t(v)$ contains only nodes of the form w^i for $i \geq 0$ and $w \in A$. Hence, paths $Q_t(v)$ and P cannot share an edge. Therefore, the invariant remains valid.

Lemma 6.4. For each $C \in \mathcal{C}$, the flow-augmenting paths in \mathcal{P}_C are pairwise edge disjoint.

Proof. We prove the statement by induction. Initially, $\mathcal{P}_C = \emptyset$ for each $C \in \mathcal{C}$, thus the base case is trivially fulfilled. For the induction step, assume, that prior to the current request the statement holds for all $C \in \mathcal{C}$.

There are two cases, where C or P_C changes: the CompMerge operation, and if a procedure creates a new path.

- Assume that the CompMerge operation merges two connected components A and B. By induction hypothesis, the paths in \mathcal{P}_A are pairwise edge disjoint. Same holds for the paths in \mathcal{P}_B . By Fact 6.2, the paths in $\mathcal{P}_A \cup \mathcal{P}_B$ are edge disjoint.
- Assume a new path P is added to \mathcal{P}_A . Clearly, there is a procedure I(A) that manages A. According to Lemma 4.3 and Lemma 6.3, the Invariant 4.2 remains valid throughout the execution of a procedure, as well as a merge of two procedures. Hence, the invariant is preserved at any time. Due to the inductive structure of the proof in Lemma 4.3, we derive that the new path P must be edge disjoint with paths in \mathcal{P}_A .

Hence, for each $C \in \mathcal{C}$, the flow-augmenting paths in \mathcal{P}_C are pairwise edge disjoint.

The previous lemmas lead to the following conclusion.

Lemma 6.5. The flow-augmenting paths in \mathcal{P} are pairwise edge disjoint.

Proof. According to Lemma 6.4 for each $C \in \mathcal{C}$, the flow-augmenting paths in \mathcal{P}_C are pairwise edge disjoint. By Lemma 6.2, for two components $A \neq B$, the flow-augmenting paths in $\mathcal{P}_A \cup \mathcal{P}_B$ are also edge disjoint. Hence, the paths in $\mathcal{P} = \bigcup_{C \in \mathcal{C}} \mathcal{P}_C$ must be pairwise edge disjoint. \square

6.5. Cost Analysis

In this section, we analyze the costs incurred by the algorithm. By design, the algorithm avoids any communication cost. Therefore, the only costs we need to consider are related to migration.

A migration may occur if a process changes its cluster or if a cluster is rescheduled to another server. We denote the costs associated with these actions as $cost_{cluster}$ and $cost_{schedule}$, respectively. Consequently, the overall cost incurred by the algorithm is at most $cost_{cluster} + cost_{schedule}$. We further decompose $cost_{cluster}$ into following subcategories:

- cost_{flow}: The cost incurred by FLOW procedures.
- cost_{large}: The cost of forming a large cluster, and merging a large cluster with other clusters.
- cost_{merge}: The cost of instance merge operations, which occur when two procedures merge. The cost involve merging special pieces, as well as pieces with the same color.
- \bullet cost_{mono}: The cost of either moving a piece to a colored cluster, or splitting a piece out of its colored cluster.

Thus, $cost_{cluster} = cost_{flow} + cost_{large} + cost_{merge} + cost_{mono}$.

Clustering Cost

Now, we will analyze the different clustering costs in detail. As a first step, we derive a bound on $cost_{flow}$.

Upper bound on $cost_{flow}$

For a connected component $C \in \mathcal{C}$, let cost(C) denote the costs incurred by all procedures for requests $\sigma_t = \{u, v\}$, where both u and v are in C.

Lemma 6.6. For all components $C \in \mathcal{C}_S$, $cost(C) \leq |F_C|^2 - |X_C|$ holds.

Proof. We prove the statement by induction. Initially, the sets F_C and X_C are empty. Hence, the base case is trivial. For the induction step, assume, that prior to the current request $\cot(C) \leq |F_C|^2 - |X_C|$ for all $C \in \mathcal{C}_S$ holds.

Due to the inductive nature of the proof of Lemma 4.6, it suffices to show that after a procedure merge, the invariant holds for the new procedure.

Assume that components $A, B \in \mathcal{C}_S$ merge into $C = A \cup B$. We show that the invariant holds for the new component C. By construction, the set of free processes in C is $F_C = F_A \cup F_B$, and the special piece is $X_C = X_A \cup X_B$. Clearly, $\operatorname{cost}(C) = \operatorname{cost}(A) + \operatorname{cost}(B)$. Therefore,

$$cost(C) = cost(A) + cost(B)
\leq |F_A|^2 - |X_A| + |F_B|^2 - |X_B|
\leq (|F_A| + |F_B|)^2 - (|X_A| + |X_B|)
= |F_C|^2 - |X_C|.$$

The second inequality holds due to the fact that $x^2 + y^2 \le (x + y)^2$, for $x, y \ge 0$. Hence, the invariant is preserved for the new component C.

Lemma 6.7. For all components $C \in \mathcal{C}_L$, $cost(C) \leq \sqrt{k}|F_C|$ holds.

Proof. We prove the statement by induction on the number of requests. Initially, the set C_L is empty, thus the base case trivially holds. For the induction step, assume that prior to the current request, $\cot(C) \leq \sqrt{k}|F_C|$ holds for all $C \in C_L$.

There are no instances that manage components $C \in \mathcal{C}_L$, and therefore no instance can generate new cost. However, there are two cases where the set \mathcal{C}_L changes. We show that in each case, the invariant is preserved.

• Component merge:

Assume, the components A and B are merged into $C = A \cup B$, where w.l.o.g. $A \in \mathcal{C}_L$. Thus the set \mathcal{C}_L changes. We show that the invariant holds for the new component C. By construction, $F_C = F_A \cup F_B$ and $\operatorname{cost}(C) = \operatorname{cost}(A) + \operatorname{cost}(B)$. By induction hypothesis, $\operatorname{cost}(A) \leq \sqrt{k}|F_A|$. Same holds for B, if $B \in \mathcal{C}_L$. Otherwise, by Lemma 6.6, $\operatorname{cost}(B) \leq |F_B|^2 - |S_B| \leq \sqrt{k}|F_B|$, since by design, the number of free processes of a procedure is at most \sqrt{k} . Hence,

$$cost(C) \le \sqrt{k}|F_A| + \sqrt{k}|F_B| = \sqrt{k}|F_C|.$$

Therefore, the invariant is preserved for the component C.

• Instance stop:

By Lemma 6.6, at the time an instance I(C) stops, $cost(C) \leq |F_C|^2 - |S_C| \leq \sqrt{k}|F_C|$ holds. Subsequently, the component C is moved to C_L . Thus, the invariant is preserved.

Applying the previous two lemmas gives:

Lemma 6.8. $cost_{flow} \leq \sqrt{k}|F|$.

Proof. According to Lemmas 6.6 and 6.7,

$$cost_{flow} = \sum_{C \in \mathcal{C}_S} cost(C) + \sum_{C \in \mathcal{C}_L} cost(C)$$

$$\leq \sum_{C \in \mathcal{C}_S} (|F_C|^2 - |S_C|) + \sum_{C \in \mathcal{C}_L} \sqrt{k} |F_C|$$

$$\leq \sqrt{k} \sum_{C \in \mathcal{C}} |F_C|$$

$$= \sqrt{k} |F|.$$

The second inequality holds because for each instance I(C), $|F_C| \leq \sqrt{k}$.

Next, we analyze $cost_{large}$.

Upper bound on $cost_{large}$

Lemma 6.9. $\operatorname{cost}_{\operatorname{large}} \leq \sqrt{k}|F|$.

Proof. We prove the statement using a charging argument. We distribute the cost among the free processes, and show that each free process is charged at most \sqrt{k} . We distinguish between two cases in which cost_{large} is incurred:

- Formation of a large cluster
 When an instance terminates and a large cluster C is formed, we mark all free processes
 in C. The cost of merging the pieces of component C into a single cluster, which is at
 most |C|, is then distributed equally among the marked free processes in C.
- Merging with a large cluster
 Suppose, the CompMerge(A, B) operation is executed, where one of the components is a
 large cluster. Let A be this component. The cost of this operation is at most |B|. We
 unmark all marked processes in B and distribute the cost of the merge equally among the
 marked free processes in A.

Now, we calculate the charge of free processes $v \in C$, where C is a large cluster. Notice that only marked processes are charged. By construction, there are at least \sqrt{k} marked free processes in C. Since $|C| \leq k$, at most k processes could have migrated into C due to the above two cases. Therefore, each marked free process is charged at most $\frac{k}{\sqrt{k}} = \sqrt{k}$ until it is unmarked. Then, $\text{cost}_{\text{large}} \leq \sum_{v \in F} \sqrt{k} = \sqrt{k} |F|$.

Next, we establish bounds on $cost_{merge}$ and $cost_{mono}$, using the charging scheme presented in Appendix A.2.

Upper bound on $cost_{merge}$

Assume, we merge two FLOW instances $I(C_A)$ and $I(C_B)$ into a combined instance $I(C_A \cup C_B)$. The merging process involves following operations:

- Merging special pieces
 The two special pieces (which by design are singleton clusters) merge into one special piece.
- Merging colored pieces
 Pieces with the same color c merge into a single c-colored piece.

The cost of the instance merge operation is defined as the number of processes that change clusters. A merge of two pieces is implemented by migrating the processes of the smaller piece to the larger piece, subsequently deleting the smaller piece. Importantly, not every merge of pieces incurs a cost.

Let $A \subseteq C_A$ and $B \subseteq C_B$ be two pieces (either special or colored) that merge. We distinguish between two types of merges between pieces:

- Monochromatic merge
 A and B belong to the same colored cluster. In this case, no cost is incurred, since no process changes its cluster.
- Non-monochromatic merge Either A or B is a singleton cluster. The processes of the smaller piece migrate to the larger piece. The cost of this merge is $\min(|A|, |B|)$.

Now, we analyze the costs of non-monochromatic merges. We develop a charging scheme to distribute these costs to the free processes in $C_A \cup C_B$. Without loss of generality, assume that A is a singleton cluster. By construction, we have that $f(v) \geq \frac{1}{4}$. For a free process v we distinguish between two types of charges: $\operatorname{charge}_f(v)$ and $\operatorname{charge}_m(v)$.

- Case 1: $\text{ffv}(B) \ge \frac{1}{16}$: W.l.o.g., let $|C_A| \le |C_B|$. The cost of this merge operation is $\min(|A|, |B|) \le 16 \min(|F_A|, |F_B|) \le 16|F_A|$. We distribute this cost by increasing $\text{charge}_f(v)$ by 16 for each $v \in F_A$.
- Case 2: $\text{ffv}(B) < \frac{1}{16}$: We charge each $v \in F_A \cup F_B$ the cost $\frac{\min(|A|,|B|)}{|F_A|+|F_B|}$.

Note, that each free process is charged at most once during an instance merge. The above two cases imply that the total $cost cost_{merge}$ is bounded by:

$$cost_{merge} \leq \sum_{v \in F} (charge_f(v) + charge_m(v))$$
.

Next, we will analyze the detailed contributions of $\operatorname{charge}_{\mathbf{f}}(v)$ and $\operatorname{charge}_{\mathbf{m}}(v)$ to the overall cost.

Lemma 6.10. For each free process v, charge_f $(v) \le 16 \log k$.

Proof. Fix a process v, and assume that $\operatorname{charge}_{\mathbf{f}}(v)$ increases during an instance merge. This increase happens only when v is part of the smaller connected component, i.e., $v \in C_A$ with $|C_A| \leq |C_B|$.

After such a merge, v belongs to the new connected component $C_A \cup C_B$ with $|C_A \cup C_B| \ge 2|C_A|$. This means that the size of the connected component containing v at least doubles each time charge_f(v) increases.

Since the size of any component is bounded by k, it can only double at most $\log k$ times before it reaches k. Therefore, $\operatorname{charge}_{\mathbf{f}}(v)$ can increase at most $\log k$ times. Given that each increase is by 16, we conclude that $\operatorname{charge}_{\mathbf{f}}(v) \leq 16 \log k$ for each free process v.

Now, we derive the contribution of $\operatorname{charge}_{\mathrm{m}}(v)$.

Lemma 6.11.
$$\sum_{v \in V} \text{charge}_{\mathbf{m}}(v) \leq 64\sqrt{k}|F|$$

Proof. Consider an iteration of a FLOW procedure. It ends only when a new flow-augmenting path is created by the procedure. Importantly, each colored cluster can only grow during an iteration.

Assume, that at the beginning of an iteration $\operatorname{charge}_{\mathrm{m}}(v)=0$ for each $v\in C$ holds. Let A be a piece maintained by the procedure. We utilize the charging scheme presented in Appendix A.2 and define

$$\operatorname{charge-avg}(A) = \frac{\sum_{v \in F_A} \operatorname{charge_m}(v)}{F_A} ,$$

as the average charge of the free processes in A.

We maintain the following invariant: For each piece A in the flow procedure, during the current iteration, the inequality charge-avg $(A) \le 64(1 - \text{ffv}(A))$ holds.

At the beginning of an iteration, charge-avg(A) = 0 for each colored cluster A, so the invariant is trivially satisfied. Additionally, each free process v in the special cluster is interpreted as a singleton piece $\{v\}$ with charge-avg($\{v\}$) = 0, satisfying the base case. For the induction step, assume that the invariant holds prior to the current request.

There are several ways in which a piece A can change:

- A free process is added to the piece A by the FLOW procedure. Since $\operatorname{charge}_{\mathbf{m}}(v)$ remains unchanged for all processes $v \in A \cup \{v\}$, we apply Lemma A.3 to sets A and $\{v\}$. Consequently, $\operatorname{charge-avg}(A \cup \{v\}) \leq 64(1 - \operatorname{flv}(A \cup \{v\}))$.
- An instance merge is executed, potentially merging A with another piece B. We distinguish between two cases:
 - charge_m(v) remains unchanged for all $v \in A \cup B$. We apply Lemma A.3 to sets A and B, ensuring charge-avg($A \cup B$) $\leq 64(1 - \text{ffv}(A \cup B))$.
 - A charge is applied to $\operatorname{charge}_{\mathrm{m}}(v)$ for some $v \in A \cup B$. This happens only when $\operatorname{ffv}(A) \geq \frac{1}{4}$ and $\operatorname{ffv}(B) \leq \frac{1}{16}$ (or vice versa). In this case, we apply Lemma A.4 to sets A and B. Hence, $\operatorname{charge-avg}(A \cup B) \leq 64(1 - \operatorname{ffv}(A \cup B))$.

From the above cases, we conclude that charge-avg $(A) \le 64(1 - \text{ffv}(A)) \le 64$ throughout the entire iteration. Thus, each free process is charged at most 64 during a single FLOW iteration, on average. Since there are at most \sqrt{k} iterations (then the procedure terminates), the total cost incurred across all iterations is:

$$\sum_{v \in F} \operatorname{charge}_{\mathbf{m}}(v) = \sum_{A} \sum_{v \in F_A} \operatorname{charge}_{\mathbf{m}}(v) = \sum_{A} \operatorname{charge-avg}(A) |F_A| \leq \sum_{A} 64 \sqrt{k} |F_A| = 64 \sqrt{k} |F| \ .$$

Finally, we can state the following lemma:

Lemma 6.12. $\operatorname{cost}_{\operatorname{merge}} \leq \mathcal{O}(\sqrt{k})|F|$.

Proof. Applying Lemma 6.10 and Lemma 6.11 gives,

$$\begin{split} & \operatorname{cost_{merge}} \leq \sum_{v \in F} (\operatorname{charge_f}(v) + \operatorname{charge_m}(v)) \\ &= \sum_{v \in F} \operatorname{charge_f}(v) + \sum_{v \in F} \operatorname{charge_m}(v) \\ &\leq 16 \log k |F| + 64 \sqrt{k} |F| \\ &\leq 80 \sqrt{k} |F| \enspace . \end{split}$$

Upper bound on $\mathrm{cost}_{\mathrm{mono}}$

The following scenarios can increase cost_{mono}:

• Case 1: At the end of an iteration of a FLOW procedure, all free processes migrate to the special piece, making all colored pieces 1-monochromatic. As a result, these colored pieces are then migrated back to their corresponding colored cluster.

- Case 2: A piece A that was previously assigned to a colored cluster becomes γ -monochromatic with $\gamma < \frac{1}{2}$. This means, the piece A now contains more than $\frac{1}{2}|A|$ free processes. Consequently, A is removed from the colored cluster and becomes a singleton cluster. The cost of this operation is |A|.
- Case 3: A piece A that was previously a singleton cluster becomes γ -monochromatic with $\gamma > \frac{3}{4}$. This implies that A now contains fewer than $\frac{1}{4}$ free processes, leading to its migration back to its corresponding colored cluster. The cost of this operation is |A|.

We now demonstrate that cost_{mono} is bounded in terms of cost_{merge} and cost_{flow}.

Lemma 6.13. $cost_{mono} \leq \mathcal{O}(cost_{merge} + cost_{flow})$.

Proof. We analyze each case separately.

- Case 1:
 - By construction, only pieces A with at least $\frac{1}{4}|A|$ free processes are not in their corresponding colored cluster at the end of an iteration. This also means, that the FLOW procedure moved at least $\frac{1}{4}|A|$ processes during the current iteration to piece A. Each such piece is moved to its corresponding colored cluster, with cost |A|. Hence, by a charging argument, the overall cost of Case 1 is at most $4 \cot A$.
- Case 2 and 3
 - First, we analyze Case 2. Consider a piece A that is either initially within its colored server, or was moved there recently. By construction, $\frac{|F_A|}{|A|} \leq \frac{1}{4}$. Suppose, that after some instance merge operations, and/or free process migrations triggered by the FLOW procedure, the instance grows in size.

Let A' be the expanded piece, and assume $|F'_A| > \frac{1}{2}|A'|$. By construction, only up to $\frac{1}{4}|A'|$ of the free processes in A' could have migrated as a result of monochromatic merges. Therefore, at least $\frac{1}{4}|A'|$ of the free processes in A' must have migrated either due to non-monochromatic merges or through the actions of the FLOW procedure. Each of these migrations contributes either to $\text{cost}_{\text{merge}}$ or $\text{cost}_{\text{flow}}$.

Hence, the piece A observed at least $\frac{1}{4}|A'|$ cost related to $cost_{merge}$ and/or $cost_{flow}$, since it was last placed in its colored cluster. On the other hand, the cost to convert A' into a singleton cluster is |A'|. Thus, by applying a charging argument, the total cost for Case 2 is bounded by $4(cost_{merge} + cost_{flow})$.

- The proof for Case 3 follows a similar approach to that of Case 2, but focuses on non-free processes instead of free processes. Consider a piece A that recently became a singleton cluster. By construction, $|F_A| \ge \frac{1}{2}|A|$. Suppose, that after some instance merge operations, the instance grows, and let A' denote the expanded piece.

Assume $|F'_A| < \frac{1}{4}|A'|$, indicating that A' migrates to its corresponding colored cluster. Since A' became a singleton cluster, at least $\frac{1}{2}|A'|$ non-free processes must have migrated to A' due to non-monochromatic merges.

Therefore, the piece A incurred at least $\frac{1}{2}|A'|$ cost related to $\operatorname{cost}_{\operatorname{merge}}$, since it became a singleton cluster. Meanwhile, the cost to migrate A' to the colored cluster is |A'|. Thus, by a charging argument, the overall cost for Case 3 is bounded by $2 \operatorname{cost}_{\operatorname{merge}}$.

From the above cases, we conclude, that $cost_{mono}$ is at most $\mathcal{O}(cost_{merge} + cost_{flow})$.

Finally, we are able to derive a bound on cost_{cluster}:

Lemma 6.14.
$$\operatorname{cost}_{\operatorname{cluster}} \leq \mathcal{O}(\sqrt{k})|\mathcal{P}|$$
.

Proof. From Lemmas 6.8,6.9,6.12, and 6.13 we conclude that $\operatorname{cost}_{\text{cluster}} \leq \mathcal{O}(\sqrt{k})|F|$. Since, by Lemma 4.5 and by construction of the instance/component merge operation, $|F| = 2|\mathcal{P}|$, the result follows directly from these bounds.

Scheduling Cost

Next, we argue that the scheduling cost due to the scheduling algorithm is at most $\mathcal{O}(\cos t_{\text{cluster}})$. The proof is almost entirely transferred from [21]. For completeness, we present it again.

Lemma 6.15 ([21]).
$$\operatorname{cost}_{\text{schedule}} \leq \mathcal{O}(\frac{1}{\epsilon}) \operatorname{cost}_{\text{cluster}}$$
.

Proof. Initially, all servers are balanced, i.e., each server holds exactly k processes. After the scheduling algorithm ends, all servers have at most 2k processes assigned. Now, if at the beginning of the scheduling algorithm, there is some server s with at least $2 + \epsilon$ processes, this means that since the last execution of the scheduling algorithm on this server, at least ϵk processes migrated there due to the clustering algorithm. Executing the scheduling algorithm on s costs at most $\mathcal{O}(k)$. Thus, by a charging argument, the overall cost of the scheduling algorithm is at most $\mathcal{O}(\frac{1}{\epsilon})$ cost_{cluster}.

Overall Cost

Lemma 6.16. The cost of the algorithm is at most $\mathcal{O}(\sqrt{k})$ OPT.

Proof. According to Lemmas 6.14 and 6.15, the overall cost of the algorithm is at most

$$\mathrm{cost}_{\mathrm{cluster}} + \mathrm{cost}_{\mathrm{schedule}} \leq \mathcal{O}(\tfrac{1}{\epsilon}\sqrt{k})|\mathcal{P}|.$$

According to Lemma 6.5, the flow-augmenting paths in \mathcal{P} are edge disjoint. Applying Lemma 4.1 yields the result.

Applying Lemmas 6.16 and 6.1 we can finally state the theorem:

Theorem 1.4. There exists a deterministic online algorithm for the general learning problem with competitive ratio $\mathcal{O}(\sqrt{k})$ and augmentation $2 + \epsilon$.

A. Omitted Proofs

A.1. Proof of Lemma 5.2

Lemma 5.2. [[13]] For any request sequence σ that obeys the learning restriction the cost LEARN(σ) is at most $\mathcal{O}(k\ell \log k)$.

Proof. Although not explicitly stated in [13], the lemma can be easily derived using the charging scheme outlined there. We explain briefly, how the cost of the LEARN algorithm in [13] is allocated. Specifically, the cost is either assigned to processes, denoted by $\operatorname{charge}(v)$ for each process v, or as an extra charge Extra.

We adapt the lemmas from [13] to align with our notation:

- Lemma 13: The total extra charge Extra is at most $\mathcal{O}(k\ell \log k)$.
- Lemma 14: The maximum process charge that a process v can receive is at most $\mathcal{O}(\log k)$.

From these two lemmas, we conclude that the total cost of the LEARN algorithm is bounded by

$$\text{LEARN} = \sum_{v} \text{charge}(v) + \text{Extra} \le \sum_{v} c \log k + c' k \ell \log k \le (c + c') k \ell \log k,$$

for some constants c and c'.

A.2. Charging Scheme for Lemma 6.12

Consider a set of processes A, with a subset $F_A \subseteq A$ representing the free processes in A. We define a charging scheme to distribute a charge among these free processes.

For each free process $v \in A$, let charge(v) denote the charge assigned to v. The total charge for the set A is the sum of the charges of all free processes in A:

$$\operatorname{charge}(A) = \sum_{v \in F_A} \operatorname{charge}(v) .$$

We then define the average charge distributed among the free processes in the set A as

$$\operatorname{charge-avg}(A) = \frac{\operatorname{charge}(A)}{|F_A|} = \frac{\sum_{v \in F_A} \operatorname{charge}(v)}{|F_A|}.$$

Next, we introduce the free process fraction of a set A, denoted by ffv(A):

$$ffv(A) = \frac{|F_A|}{|A|} .$$

Furthermore, for two disjoint sets A and B, we define the following weighted averages:

size-avg_{A,B}(X,Y) =
$$\frac{|A|}{|A|+|B|}X + \frac{|B|}{|A|+|B|}Y$$
 and free-avg_{A,B}(X,Y) = $\frac{|F_A|}{|F_A|+|F_B|}X + \frac{|F_B|}{|F_A|+|F_B|}Y$.

For simplicity, we may omit the subscripts A, B when the sets are clear from the context. Clearly, we have $\min\{X,Y\} \leq \text{size-avg}(X,Y) \leq \max\{X,Y\}$ and $\min\{X,Y\} \leq \text{free-avg}(X,Y) \leq \max\{X,Y\}$ as this holds for any weighted average. The following claim essentially states that for $\text{ffv}(A) \leq \text{ffv}(B)$ the function free-avg tends more towards the second parameter (when compared to size-avg).

Fact A.1. Let $X \ge Y$ and $\text{ffv}(A) \le \text{ffv}(B)$. Then

$$free-avg(X,Y) \le size-avg(X,Y)$$
.

Proof. Both functions are weighted averages, i.e., functions of the form $(1-\lambda)X+\lambda Y$. Therefore, it is sufficient to show that the multiplier for the second parameter is larger in the function free-avg. Observe that

$$\lambda_{\text{free}} = \frac{|F_B|}{|F_A| + |F_B|} = \frac{|B| \cdot \text{ffv}(B)}{|A| \cdot \text{ffv}(A) + |B| \cdot \text{ffv}(B)} \ge \frac{|B| \cdot \text{ffv}(B)}{|A| \cdot \text{ffv}(B) + |B| \cdot \text{ffv}(B)} = \frac{|B|}{|A| + |B|} = \lambda_{\text{size}} \ ,$$

where the inequality holds because $ffv(A) \leq ffv(B)$.

Fact A.2. Suppose that we merge two sets A and B into $A \cup B$. Then

$$\operatorname{charge-avg}(A \cup B) = \operatorname{free-avg}(\operatorname{charge-avg}(A), \operatorname{charge-avg}(B)) \quad and \quad \operatorname{ffv}(A \cup B) = \operatorname{size-avg}(\operatorname{ffv}(A), \operatorname{ffv}(B)) \ .$$

Proof. The total charge in $A \cup B$ is charge-avg $(A) \cdot |F_A|$ + charge-avg $(B) \cdot |F_B|$. Dividing this by the number $|F_A| + |F_B|$ of free processes in $A \cup B$ gives the first equality. The total number of free processes in $A \cup B$ is ffv $(A) \cdot |A|$ + ffv $(B) \cdot |B|$. Dividing this by the total number |A| + |B| of processes in $A \cup B$ gives the second equality.

Lemma A.3. Suppose we are given two sets A, B that fulfill

charge-avg
$$(A) \le c(1 - \text{ffv}(A))$$
 and charge-avg $(B) \le c(1 - \text{ffv}(B))$

for a constant c > 0. Then, charge-avg $(A \cup B) \le c(1 - \text{ffv}(A \cup B))$.

Proof. W.l.o.g. assume that $ffv(A) \leq ffv(B)$. We differentiate between two cases:

• $\operatorname{charge-avg}(A) \leq \operatorname{charge-avg}(B)$

$$\operatorname{charge-avg}(A \cup B) = \operatorname{free-avg}(\operatorname{charge-avg}(A), \operatorname{charge-avg}(B))$$

$$\leq \operatorname{charge-avg}(B)$$

$$\leq c(1 - \operatorname{ffv}(B))$$

$$\leq c(1 - \operatorname{size-avg}(\operatorname{ffv}(A), \operatorname{ffv}(B)))$$

$$= c(1 - \operatorname{ffv}(A \cup B))$$

• $\operatorname{charge-avg}(A) \ge \operatorname{charge-avg}(B)$

$$\begin{split} \operatorname{charge-avg}(A \cup B) &= \operatorname{free-avg}(\operatorname{charge-avg}(A), \operatorname{charge-avg}(B)) \\ &\leq \operatorname{size-avg}(\operatorname{charge-avg}(A), \operatorname{charge-avg}(B)) \\ &\leq \operatorname{size-avg}(c(1 - \operatorname{ffv}(A)), c(1 - \operatorname{ffv}(B))) \\ &= c(1 - \operatorname{size-avg}(\operatorname{ffv}(A), \operatorname{ffv}(B))) \\ &= c(1 - \operatorname{ffv}(A \cup B)) \end{split}$$

Lemma A.4. Suppose we are given two sets A, B with $\text{ffv}(A) \geq \frac{1}{4}$ and $\text{ffv}(B) \leq \frac{1}{16}$ that fulfill

charge-avg(A)
$$\leq 64(1 - \text{ffv}(A))$$
 and charge-avg(B) $\leq 64(1 - \text{ffv}(B))$.

Assume a merge of A and B is executed, where the free processes $F_A \cup F_B$ are charged with cost $\min(|A|, |B|)$. Then, after the merge, charge-avg $(A \cup B) \le 64(1 - \text{ffv}(A \cup B))$.

Proof.

$$\operatorname{charge-avg}(A \cup B) = \operatorname{free-avg}(\operatorname{charge-avg}(A), \operatorname{charge-avg}(B)) + \frac{\min(|A|, |B|)}{|F_A| + |F_B|}$$

$$\leq \operatorname{free-avg}(64(1 - \operatorname{ffv}(A)), 64(1 - \operatorname{ffv}(B))) + \frac{\min(|A|, |B|)}{|F_A| + |F_B|}$$

$$= 64(1 - \operatorname{free-avg}(\operatorname{ffv}(A), \operatorname{ffv}(B))) + \frac{\min(|A|, |B|)}{|F_A| + |F_B|}$$

Furthermore,

$$\begin{aligned} \text{free-avg}(\text{ffv}(A), \text{ffv}(B)) - \text{ffv}(A \cup B) &= \frac{|F_A|^2}{|A|(|F_A| + |F_B|)} + \frac{|F_B|^2}{|B|(|F_A| + |F_B|)} - \frac{|F_A| + |F_B|}{|A| + |B|} \\ &= \frac{|F_A|^2 |B| + |F_B|^2 |A|}{|A||B|(|F_A| + |F_B|)} - \frac{|F_A| + |F_B|}{|A| + |B|} \\ &= \frac{(|B||F_A| - |A||F_B|)^2}{|A||B|(|A| + |B|)(|F_A| + |F_B|)} \end{aligned}$$

Let $g(x) = |B||F_A| - |A|x$. For $0 \le x < \frac{|F_A|}{|A|}|B|$, the function is positive and monotonically decreasing. Hence, for $x \le \frac{1}{16}|B| < \frac{|F_A|}{|A|}|B|$ the function attains its minimum at $x = \frac{1}{16}|B|$. Therefore, $g(\frac{1}{16}|B|) = |B||F_A| - \frac{1}{16}|A||B| \ge \frac{1}{4}|A||B| - \frac{1}{16}|A||B| = \frac{3}{16}|A||B| > 0$. Thus, $(g(x))^2 \ge \frac{9}{16^2}|A|^2|B|^2 > \frac{1}{32}|A|^2|B|^2$, for $x \le \frac{1}{16}|B|$. Hence,

$$\frac{(|B||F_A| - |A||F_B|)^2}{|A||B|(|A| + |B|)(|F_A| + |F_B|)} \ge \frac{1}{32} \frac{|A|^2|B|^2}{|A||B|(|A| + |B|)(|F_A| + |F_B|)}$$

$$= \frac{1}{32} \frac{|A||B|}{(|A| + |B|)(|F_A| + |F_B|)}$$

$$\ge \frac{1}{64} \frac{\min(|A|, |B|) \cdot \max(|A||B|)}{\max(|A||B|) \cdot (|F_A| + |F_B|)}$$

$$= \frac{1}{64} \frac{\min(|A|, |B|)}{|F_A| + |F_B|}.$$

The second inequality holds because $|A| + |B| \le 2 \max(|A|, |B|)$. Therefore,

$$\begin{aligned} \operatorname{charge-avg}(A \cup B) &\leq 64(1 - \operatorname{free-avg}(\operatorname{ffv}(A), \operatorname{ffv}(B))) + \frac{\min(|A|, |B|)}{|F_A| + |F_B|} \\ &\leq 64(1 - \operatorname{free-avg}(\operatorname{ffv}(A), \operatorname{ffv}(B))) + 64(\operatorname{free-avg}(\operatorname{ffv}(A), \operatorname{ffv}(B)) - \operatorname{ffv}(A \cup B)) \\ &= 64(1 - \operatorname{ffv}(A \cup B)). \end{aligned}$$

B. Overview for Deterministic Online Algorithms

The following table provides a summary of the current results in the context of the online balanced partitioning problem. The results are categorized across three different models: the General Model (GM), the Learning Model (LM), and the Generalized Learning Model (GLM).

Model	Augmentation	Upper Bounds	Lower Bounds
GM	$2 + \epsilon$	$\mathcal{O}(k \log k)$	$\Omega(k)$
	$1 + \epsilon$	$\mathcal{O}(k\ell \log k)$	$\Omega(\max(k, \ell \log k))$
	1	$\mathcal{O}((k\ell)^2)$	$\Omega(k\ell)$
LM	$2 + \epsilon$	$\mathcal{O}(\log k)$	$\Omega(\log k)$
	$1 + \epsilon$	$\mathcal{O}(\ell \log k)$	$\Omega(\ell \log k)$
	1	$\mathcal{O}(k\ell)$	$\Omega(k\ell)$
GLM	$2 + \epsilon$	$\mathcal{O}(\sqrt{k})$	$\Omega(\sqrt{k})$
	$1 + \epsilon$	$\mathcal{O}(\max(\sqrt{k\ell\log k}, \ell\log k))$	$\Omega(\max(\sqrt{k\ell\log k}, \ell\log k))$
	1	$\mathcal{O}((k\ell)^2)$	$\Omega((k\ell))$

References

- [1] Behnaz Arzani, Siva Kesava Reddy Kakarla, Miguel Castro, Srikanth Kandula, Saeed Maleki, and Luke Marshall. Rethinking machine learning collective communication as a multi-commodity flow problem. arXiv preprint arXiv:2305.13479, 2023.
- [2] Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic balanced graph partitioning. In SIAM J. Discrete Math (SIDMA), 2019.
- [3] Chen Avin, Louis Cohen, Mahmoud Parham, and Stefan Schmid. Competitive clustering of stochastic communication patterns on a ring. In *Journal of Computing*, 2018.
- [4] Chen Avin, Louis Cohen, Mahmoud Parham, and Stefan Schmid. Competitive clustering of stochastic communication patterns on a ring. In *Journal of Computing*, 2018.
- [5] Chen Avin, Louis Cohen, and Stefan Schmid. Competitive clustering of stochastic communication patterns on the ring. In *Proc. 5th International Conference on Networked Systems (NETYS)*, 2017.
- [6] Chen Avin, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Online balanced repartitioning. In *Proc. 30th International Symposium on Distributed Computing (DISC)*, 2016.
- [7] Yossi Azar, Chay Machluf, Boaz Patt-Shamir, and Noam Touitou. Competitive vertex recoloring. In 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022). Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.
- [8] Marcin Bienkowski, Martin Böhm, Martin Kouteckỳ, Thomas Rothvoß, Jiří Sgall, and Pavel Veselỳ. Improved analysis of online balanced clustering. In *International Workshop on Approximation and Online Algorithms*, pages 224–233. Springer, 2021.
- [9] Marcin Bienkowski and Stefan Schmid. A subquadratic bound for online bisection. In 41st International Symposium on Theoretical Aspects of Computer Science (STACS), 2024.
- [10] Björn Feldkord, Matthias Feldotto, Anupam Gupta, Guru Guruganesh, Amit Kumar, Sören Riechers, and David Wajc. Fully-dynamic bin packing with little repacking. In *ICALP*, pages 51:1–51:24, 2018.
- [11] Tobias Forner, Harald Raecke, and Stefan Schmid. Online balanced repartitioning of dynamic communication patterns in polynomial time. In *Proc. SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021.

- [12] Chen Griner, Johannes Zerwas, Andreas Blenk, Manya Ghobadi, Stefan Schmid, and Chen Avin. Cerberus: The power of choices in datacenter topology design-a throughput perspective. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(3):1–33, 2021.
- [13] Monika Henzinger, Stefan Neumann, Harald Raecke, and Stefan Schmid. Tight bounds for online graph partitioning. In *Proc. ACM-SIAM Symposium on Discrete Algorithms* (SODA), 2021.
- [14] Monika Henzinger, Stefan Neumann, and Stefan Schmid. Efficient distributed workload (re-)embedding. In *Proc. ACM SIGMETRICS*, 2019.
- [15] Dorit S Hochbaum and David B Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.
- [16] Felix Hohne, Soren Schmitt, and Rob van Stee. Algorithms column 38: 2021 in review. SIGACT News 52, 2021.
- [17] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pages 267–280, 2017.
- [18] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. ACM SIGCOMM Computer Communication Review, 42(5):44–48, 2012.
- [19] Maciej Pacut, Mahmoud Parham, and Stefan Schmid. Optimal online balanced graph partitioning. In *Proc. IEEE INFOCOM*, 2021.
- [20] Harald Räcke, Stefan Schmid, and Ruslan Zabrodin. Approximate dynamic balanced graph partitioning. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 401–409, 2022.
- [21] Harald Räcke, Stefan Schmid, and Ruslan Zabrodin. Polylog-competitive algorithms for dynamic balanced graph partitioning for ring demands. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 403–413, 2023.
- [22] Rajmohan Rajaraman. Improved bounds for online balanced graph re-partitioning. In 30th Annual European Symposium on Algorithms (ESA 2022), 2022.
- [23] Rajmohan Rajaraman and Omer Wasim. Competitive capacitated online recoloring. arXiv preprint arXiv:2408.05370, 2024.
- [24] Peter Sanders, Naveen Sivadasan, and Martin Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009.